Universitat de Lleida

Escola Politècnica Superior

Enginyería Tècnica en Informàtica de Gestió

Treball de final de carrera

APORTACIÓN AL ESTUDIO DE LOS DIGRAFOS RADIALES DE MOORE

Autora: Cristina Huerta Aguilar
Director: Nacho López Lorenzo
Julio, 2009

Índice

Índice de figuras

1. Introducción, objetivos, y estructura de la memoria

En este apartado introductorio se ha incluido una descripción general de los objetivos básicos que persigue el programa sobre el cual está basado este proyecto y también tenemos un apartado donde se muestra la estructura que seguirá esta documentación junto con una breve explicación sobre su contenido.

1.1. Introducción

Como en todas las ciencias, se han desarrollado de forma extensa muchas áreas pero este proyecto está basado en una de ellas que es la Teoría de Grafos, la cual abarca una rama muy extensa del área de las matemáticas y concretamente de las Ciencias de la Computación. Sus aplicaciones son bastante diversas, pero nosotros nos vamos a centrar principalmente en las que tengan relación directa con este proyecto o nos ayuden a comprender algún concepto. Los grafos son unas estructuras que permiten crear modelos de ejemplos comunes que se encuentran muy a menudo en la vida real, como por ejemplo, el cálculo del camino mínimo entre dos ciudades determinadas, la creación de redes informáticas con unas características especificas, etc.

Para poder entender el comportamiento de los grafos es necesario entender sus propiedades principales, aprender algunas definiciones básicas y conocer algunos de los algoritmos (los más utilizados) que calculan las distancias entre cada par de vértices.

1.2. Objetivos del proyecto

Antes de definir y explicar conceptos detallados orientados a este trabajo es necesario conocer los resultados que se desean obtener y en este punto están comentados de forma general los objetivos que se habrán intentado resolver al finalizar el proyecto. La finalidad del mismo es la creación de un programa en C++ que, dadas unas características iniciales asociadas a un digrafo radial de Moore específico, se obtengan todos los digrafos radiales de Moore, de los cuales se calcularán una serie de características: la matriz de excentricidades, sus secuencias de excentricidades (tanto de entrada como de salida), la secuencia de excentricidades resultante de las dos anteriores, el número de vértices centrales del digrafo... Cada digrafo resultante se guardará en un archivo en un formato que sea compatible con el Mathematica. Hay varios

tipos de formas de representar los digrafos pero se ha optado por una de ellas que es la representación del digrafo mediante su matriz de adyacencia.

1.3. Estructura de la memoria

La memoria se encuentra dividida en una serie de puntos que contendrán la siguiente información:

1.3.1. Introducción, objetivos y estructura de la memoria

Explica de forma resumida en que consistirá el proyecto, los objetivos que se intentaran resolver al realizar el mismo y la estructura que mantendrá la memoria.

1.3.2. Conceptos básicos sobre digrafos

Contendrá una serie de definiciones necesarias para comprender la totalidad del proyecto y el entendimiento de conceptos básicos más complejos relacionados de forma directa con el funcionamiento del mismo.

1.3.3. Recorridos, conexión y distancias en un digrafo

Están explicadas detalladamente una serie de definiciones relacionadas con el cálculo de distancias. También incluye los algoritmos necesarios para calcular las excentricidades y explica su funcionamiento.

1.3.4. Digrafos radiales de Moore

Debido a que este proyecto se basa en este tipo de digrafos, definiremos formalmente y explicaremos de forma precisa que características tienen estas estructuras.

1.3.5. Estructura base

Los digrafos radiales de Moore tienen siempre una estructura subyacente fija. Aquí está explicada la creación de esta estructura y sus características.

1.3.6. Generación de digrafos

Para generar los digrafos radiales de Moore lo haremos por pasos, o funciones, que nos calcularán todas las posibles combinaciones de uniones entre los vértices de ese digrafo y como obtener cada posible candidato a solución.

Aquí explicamos en que consiste cada uno de los pasos y cual es el resultado que se obtendrá.

1.3.7. Isomorfismo de los digrafos

Se han realizado varias pruebas del cálculo del isomorfismo con diferentes herramientas. Para explicar dichas técnicas se han debido definir una serie de conceptos básicos del isomorfismo que se explican en este apartado.

1.3.8. Cálculo del isomorfismo empleando certificados

Se nombra la técnica que se ha utilizado para el cálculo del isomorfismo y la bibliografía donde se explica el funcionamiento de la misma. No se profundiza sobre este punto debido a que no es nuestro objetivo conocer la implementación de la herramienta pero sí que están explicados los problemas con los que nos hemos encontrado durante su utilización.

1.3.9. NAUTY

Esta es la otra herramienta utilizada para el cálculo del isomorfismo. Hay una breve explicación de su uso y funcionamiento junto con los problemas encontrados al utilizarla.

1.3.10. Guardar las combinaciones

Están explicadas las diversas variantes que se han utilizado para guardar todos los posibles resultados de los cuales se comprobará posteriormente si son o no radiales de Moore y se enuncian los problemas de las mismas.

1.3.11. Implementación del programa

Explicación del lenguaje utilizado para el desarrollo del programa y una descripción de como está estructurado y el funcionamiento de cada una de esas partes.

1.3.12. Conclusiones y trabajos futuros

Se encuentran las conclusiones obtenidas en la realización de este proyecto junto con los trabajos que pueden desarrollarse a partir de este.

2. Conceptos básicos sobre digrafos

Este capítulo es el encargado de ofrecer un conjunto de definiciones básicas ligadas al desarrollo de nuestra aplicación y que permiten ofrecer una visión general sobre las estructuras sobre las que se basa el trabajo y las propiedades que se van a tener en cuenta en todo momento en la construcción del proyecto. Se definen un conjunto de conceptos básicos sobre digrafos que nos ayudarán a comenzar a entender de que trata este proyecto [4] y [3]. No entraremos en definiciones propias de digrafos ya que no se va a trabajar con grafos no dirigidos en este trabajo.

2.1. Definición de digrafo

Un digrafo $G = (V, A)$ está formado por un conjunto finito y no vacío V y por un conjunto A de pares ordenados de elementos diferentes de V. Denominamos vértices a los elementos de V y arcos a los de A.

Si $a = (u, v)$ es un arco, entonces decimos que u es adyacente hacia v o que v es adyacente desde u. Otra forma de decirlo es que el arco $a = (u, v)$ es incidente desde u hacia v. Normalmente, para abreviar la notación se escribe $a = uv$ en lugar de $a = (u, v)$.

Definición: El orden de un digrafo $G = (V, A)$ es el número de vértices de G, es decir, el cardinal de V que denotaremos por $|V|$. La medida de G es el número de arcos de G, es decir, el cardinal de A que se denota por $|A|$.

La utilización de un digrafo, como modelo matemático, resulta más adecuado que los grafos en las situaciones donde el orden (o la dirección) de las relaciones entre los objetos es relevante. En nuestro caso se estudian solamente los grafos dirigidos.

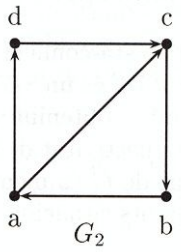

Figura 1: Representación gráfica de un digrafo G_2, cuyo conjunto de vértices es $V = \{a, b, c, d\}$ y arcos $A = \{ad, dc, ac, cb, ba\}$

2.2. Grado de un vértice

Definición: Si v es un vértice de un digrafo $G = (V, A)$, el grado de salida de v en el digrafo G, cuya notación será $g^+(v)$, es el número de arcos incidentes desde v, es decir,

$$g^+(v) = |\{u \in V \mid (v, u) \in A\}|.$$

Por el contrario, el grado de entrada de v, que será denotado por $g^-(v)$, es el número de arcos incidentes hacia v, es decir,

$$g^-(v) = |\{u \in V \mid (u, v) \in A\}|.$$

Hay una relación entre el orden, la medida y los grados de entrada y salida de los vértices de un digrafo que viene dada por el siguiente teorema

Teorema: La suma de todos los grados de salida de los vértices de un digrafo $G = (V, A)$ es igual a la suma de todos los grados de entrada y es igual a la medida del digrafo. Es decir,

$$\sum_{v \in V} g^-(v) = \sum_{v \in V} g^+(v) = |A|.$$

Definición: Un digrafo $G = (V, A)$ es simétrico si

$$\forall u, v \in V \; u \neq v, \quad (u, v) \in A \iff (v, u) \in A.$$

Un digrafo $G = (V, A)$ es asimétrico si

$$\forall u, v \in V \; u \neq v, \quad (u, v) \in A \implies (v, u) \notin A,$$

o, de manera idéntica, no hay ningún par de vértices diferentes u y v tales que $(u, v) \in A$ y $(v, u) \in A$.

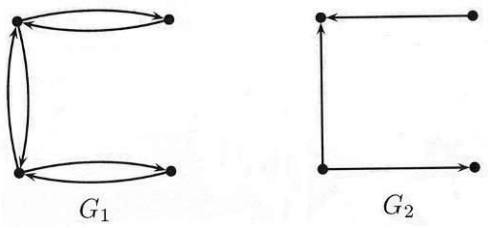

Figura 2: G_1 es un digrafo simétrico y G_2 es un digrafo asimétrico

Definición: Un digrafo $G = (V, A)$ es $d - regular$ si para todos los vértices de G, tanto el grado de entrada como de salida es igual a d.

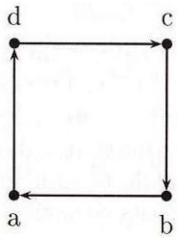

Figura 3: Representación gráfica de un digrafo $1 - regular$

Definición: En un digrafo $G = (V, A)$, un lazo es una adyacencia de un vértice consigo mismo, es decir $a = (u, u)$.

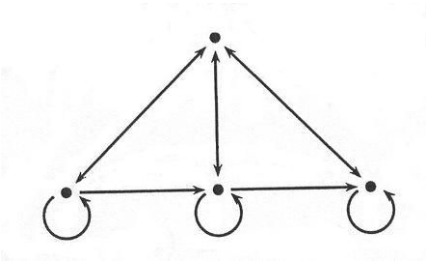

Figura 4: Representación gráfica de un digrafo con tres lazos

2.3. Representación de un digrafo

Si nos encontramos en un ámbito matemático no resulta demasiado adecuado realizar la representación de los digrafos como un par (V, A) de conjuntos finitos donde los elementos de A son de la forma (u, v), siendo u y v vértices diferentes de V ni mediante la representación por medio de un dibujo ya que no pueden ser procesadas por un ordenador. Por ello definiremos diferentes formas que existen para su representación pueda realizarse por medio de un computador que nos permita poder operar con ellos de forma sencilla, rápida y eficiente [4] y [10]. Esas representaciones serán la matriz de adyacencia y la lista de adyacencias que son las formas más comunes y sencillas para operar con estructuras de este tipo.

2.3.1. Matriz de adyacencia

Las definiciones que vienen a continuación ayudan a entender en que consiste este tipo de representación y su funcionamiento. También se comentan una serie de características que se deben tener en cuenta para entender el funcionamiento interno del proyecto.

Definición: La matriz de adyacencia de un digrafo $G = (V, A)$, con un conjunto de vértices $V = v_1, v_2, ..., v_n$, es la matriz cuadrada $A(G) = (a_{ij})$ de medida $n \times n$ definida de la forma siguiente:

$$a_{ij} = \begin{cases} 1 & \text{si } v_i \text{ es adyacente hacia } v_j \\ 0 & \text{en caso contrario} \end{cases}$$

La matriz de adyacencia de un digrafo no tiene por qué ser simétrica, sin embargo preserva los elementos nulos en la diagonal en el caso de que no haya lazos en el digrafo, es decir, que no haya una arista que tenga el mismo

vértice como origen y destino. En el caso de que hubiera lazos en cada vértice, los valores de la diagonal serían 1.

Como se ha explicado en el apartado anterior, en un digrafo tenemos grado de entrada y grado de salida. Esto se debe reflejar a la hora de su representación mediante matriz de adyacencia. Por lo tanto, el número de elementos iguales a 1 de la fila i de $E(G)$ es $g^+(v_i)$, el grado de salida del vértice v_i, mientras que el número de elementos iguales a 1 de la columna i de $E(G)$ es $g^-(v_i)$, el grado de entrada del vértice v_i.

Matriz de adyacencia del digrafo de la Figura 1:

$$M_A(G_2) = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Un buen ejemplo de lo enunciado respecto a los grados de salida y entrada en las matrices de adyacencia es coger la matriz anterior viendo que el grado de salida del vértice a es $g^+(a) = 2$ y el grado de entrada del mismo vértice $g^-(a) = 1$.

La matriz de adyacencia se puede representar de forma muy sencilla. En nuestro caso, al trabajar con la programación orientada a objetos, representaremos la matriz binaria como una clase, con sus principales atributos y operaciones. Hay un apartado específico para explicar las clases y sus operaciones por lo que no profundizaremos en ello ahora.

2.3.2. Lista de adyacencias

La matriz de adyacencia tiene un pequeño problema relacionado con el almacenamiento y es su tamaño. Este problema queda solucionado con la representación mediante la lista de adyacencias. En este proyecto no se ha utilizado esta forma de representación en el resultado final. Inicialmente se quiso utilizar para la realización de un conjunto específico de operaciones, pero se decidió no hacerlo debido a una serie de problemas a la hora de la representación que serán enumerados en un apartado posterior.

Definición: Dado el digrafo $G = (V, A)$ con el conjunto de vértices

$$V = \{v_1, v_2...., v_n\}$$

la lista de adyacencias de ese digrafo es una lista formada por n sublistas, una para cada vértice v_i, donde figuran los vértices adyacentes al correspondiente vértice v_i.

Lista de adyacencias del digrafo de la Figura 1:

$$\{\{c, d\}, \{a\}, \{b\}, \{c\}\}$$

Para la representación de un digrafo mediante listas de adyacencias se ha creado una clase Lista donde el contenido puede ser cualquier tipo de elemento (lista, entero...). En esta Lista nosotros guardaremos las sub-listas de las adyacencias de cada uno de los vértices.

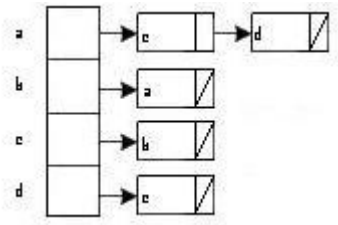

Figura 5: Lista de adyacencias del digrafo de la Figura 1

3. Recorridos, conexión y distancias en un digrafo

A continuación se definirán una serie de conceptos necesarios para poder definir y desarrollar los digrafos radiales de Moore (que están explicados más adelante) y enunciar sus correspondientes características. También aportaremos un conjunto de algoritmos que han sido utilizados en este trabajo y que han servido para calcular una serie de valores básicos para intentar resolver los objetivos que se han planteado inicialmente [4] y [2].

3.1. Introducción

Al crear un digrafo debemos tener en cuenta una serie de definiciones que nos aportarán las características principales de la estructura. En cualquier red de comunicación se deben tener en consideración esta serie de propiedades para diseñarla de una forma óptima y sin errores de conexión entre nodos.

3.2. Tipos de recorridos y conexión

En un digrafo $G = (V, A)$ pueden existir diferentes tipos de recorridos entre dos vértices u y v pero solamente nos centraremos en aquellos que sean de interés para poder profundizar en los digrafos radiales de Moore.

Definición: Un $u - v$ recorrido ($u - v$ walk) de longitud l del digrafo G es una secuencia de vértices $u_0u_1...u_{l-1}u_l$, con $u_0 = u$, $u_l = v$ y $u_{i-1}u_i \in A$ para $i = 1, 2, ...l$, es decir, cada par de vértices consecutivos son adyacentes. Si $u = v$ diremos que es un recorrido cerrado y si $u \neq v$ diremos que es un recorrido abierto.

Definición: Un camino (path) es un recorrido en el cual no se repiten los vértices.

Dado el digrafo de la Figura 6 enunciaremos un conjunto de ejemplos de las definiciones dadas de recorridos:

1541 es un 1-1 recorrido cerrado
15325 es un 1-5 recorrido abierto
1532 es un 1-2 camino

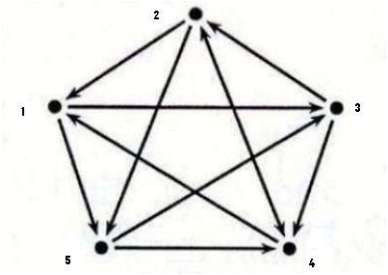

Figura 6: Digrafo G utilizado en ejemplos de tipos de recorridos

Un digrafo $G = (V, A)$ es fuertemente conexo si para todo par de vértices u y v de G existe un $u - v$ camino. Un digrafo G será débilmente conexo si el digrafo subyacente es conexo.

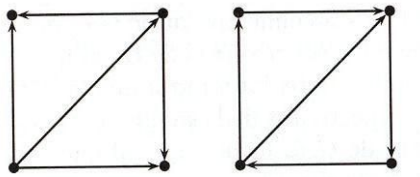

Figura 7: Digrafos débilmente conexo y fuertemente conexo respectivamente

3.3. Distancias en un digrafo

3.3.1. Excentricidades

En digrafos fuertemente conexos tiene sentido hablar del cálculo de distancias mínimas entre un vértice v y otro vértice u. En la vida real, minimizar distancias entre puntos es uno de los factores principales para situar un determinado recurso (hospital, oficina,...), y el conjunto de distancias se obtiene mediante la utilización de un algoritmo específico que enunciaremos posteriormente [4] y [8]. La idea de este proyecto es conseguir que las distancias entre los vértices estén comprendidos dentro de una cota. Esta cota la explicaremos más adelante.

Cuando hablamos de los digrafos fuertemente conexos, decimos que la distancia entre dos vértices u y v de G, que denotaremos por $d(u, v)$, es la mínima de las longitudes de todos los $u - v$ caminos de G.

La excentricidad de un vértice v de G, denotada por $e(v)$, es la distancia más grande entre v y el resto de vértices de G. En los digrafos siempre nos encontramos con dos tipos de excentricidades: la de salida y la de entrada, denotadas respectivamente por $e^+(v)$ y $e^-(v)$. Un vértice u es un vértice excéntrico del vértice v si la distancia de v a u es igual a $e(u)$.

La excentricidad de salida de un vértice v, $e^+(v)$ es la mayor de las distancias de ese vértice al resto de nodos del digrafo, es decir,

$$e^+(v) = \text{máx}\{d(v,u), u \in V\}$$

La excentricidad de entrada de un vértice v, $e^-(v)$ es la mayor de las distancias de todos los vértices del digrafo al mismo, es decir,

$$e^-(v) = \text{máx}\{d(u,v), u \in V\}$$

Como se ha comentado, un vértice de un digrafo tiene excentricidades de entrada y salida. M. Knor da una definición diferente a la que le dan otros autores en otros trabajos sobre grafos y digrafos a la excentricidad y es la siguiente:

$$e(v) = \text{máx}\{e^+(v), e^-(v)\}$$

En un apartado posterior se definen y explican detalladamente las propiedades y características de los digrafos radiales de Moore.

Definición: El diámetro de un digrafo $G = (V, A)$, denotado por $D(G)$, es la máxima de las excentricidades de G, es decir,

$$D(G) = \text{máx}\{e(v)|v \in V\}.$$

Definición: El radio de un digrafo $G = (V, A)$ denotado por $r(G)$, es la más pequeña de las excentricidades de G, es decir,

$$r(G) = \text{mín}\{e(v)|v \in V\}.$$

3.3.2. Algoritmo BFS

Durante la programación deberemos calcular la matriz de distanias del digrafo. Será el primer paso que debemos realizar para determinar si el digrafo es o no radial de Moore. Para esto, debemos utilizar el algoritmo BFS (Breadth First Search) o algoritmo de búsqueda por anchura prioritaria. Esta estrategia se caracteriza por dar preferencia, en cada paso, a la exploración

de todas las "posibles vias"desde el vértice en el que se encuentra. Este algoritmo será utilizado reiteradamente para calcular las distancias entre todos los vértices del digrafo y así obtener la matriz de distancias.

Este algoritmo realiza la exploración de tal manera que a partir de un vértice v, visita uno tras otro todos sus nodos vecinos y seguidamente todos los vecinos de sus vecinos sino han sido visitados con anterioridad y así sucesivamente hasta haber visitado todos los vértices del digrafo.

Algoritmo BFS

1: $distancia[u] \longleftarrow 0$
2: $l_{aux} \longleftarrow V - u$
3: $c \longleftarrow u$
4: mientras $(\neg vacia(c))$
5: $\quad u \longleftarrow primero(c)$
6: $\quad$ para cada $((w \in l_{aux})$ y $(w$ *adyacente con* $v))$ hacer
7: $\quad\quad distancia[w] \longleftarrow distancia[v] + 1$
8: $\quad\quad c \longleftarrow insertar(w)$
9: $\quad\quad l_{aux} \longleftarrow l_{aux} - w$
10: $\quad$ fin para cada
11: $\quad c \longleftarrow eliminar()$
12: fin mientras

4. Digrafos Radiales de Moore

4.1. Introducción

En el momento de diseñar cualquier tipo de red de comunicación hay una serie de propiedades importantes que se deben tener en cuenta para un buen funcionamiento de la red. Estas propiedades son:

- Las conexiones que tiene cualquier nodo de la estructura son limitadas.
- Para acceder de un nodo a otro, es preferible utilizar el mínimo número de nodos intermedios posible.
- Se debe maximizar el número de nodos que existen en la red de comunicación.

Estas tres propiedades son las que se estudian en El problema del grado-diámetro, punto muy importante dentro de la Teoría de Grafos. [3], [8] y [9].

Una serie de autores han demostrado que existen muy pocos casos de digrafos de Moore, por lo que se han relajado algunas condiciones para obtener más resultados y eso ha dado lugar a la obtención de los digrafos radiales de Moore, que serán los que se estudian en este proyecto.

4.2. Digrafos de Moore

Inicialmente se ofrecen unos conceptos que definen los digrafos de Moore que son de gran importancia para explicar los digrafos radiales de Moore.

Un problema que que se tiene al diseñar un digrafo es el comentado antes: el problema del grado-diámetro que consiste en encontrar el número máximo de vértices que puede tener un digrafo con grado de salida máximo d y diámetro k y la obtención de digrafos óptimos con este orden máximo.

El número de vértices para un digrafo G con grado máximo de salida d y diámetro k está acotado por la siguiente cantidad:

$$n_{d,k} = 1 + d + d^2 + ... + d^k$$

que se obtiene contando el número de vértices que aparecen en cada nivel de la Figura 8.

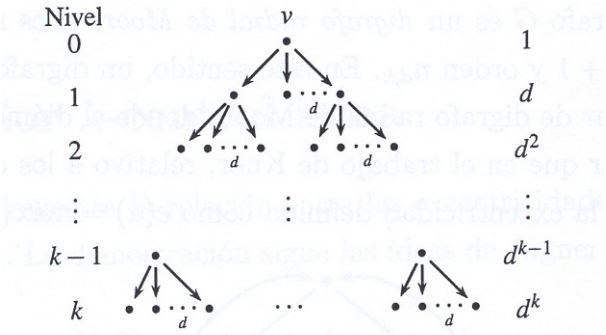

Figura 8: Número máximo de vértices en un digrafo con grado máximo de salida d y diámetro k

El valor $n_{d,k}$ se denomina cota de Moore para digrafos y los digrafos para los que se alcanza dicha cota se llaman digrafos de Moore.

Ha sido demostrado por Plesník y Znám que no existen digrafos de Moore distintos de los casos triviales $d = 1$ y $k = 1$. Como se pueden obtener muy pocos casos particulares de digrafos de Moore se ha optado por buscar estructuras aproximadas que se parezcan a estos digrafos. Una de estas opciones de aproximación para su construcción (y en la que nos vamos a basar) consiste en debilitar la condición de que todos los vértices del digrafo tengan la misma excentricidad. Esta condición dará lugar a un tipo de digrafos llamados digrafos radiales de Moore.

4.3. Digrafos radiales de Moore

Ahora que ya tenemos definido un digrafo de Moore y sabemos que la procedencia de los digrafos radiales de Moore viene de una aproximación a los digrafos de Moore, vamos a definirlos formalmente.

Definición: Se dice que un digrafo G es un digrafo radial de Moore si es regular de grado d, radio k, diámetro $\leq k + 1$ y orden $n_{d,k}$. Un digrafo de Moore sería un caso particular de un digrafo radial de Moore donde diámetro y radio coinciden.

Antes se ha dicho que solamente existen digrafos de Moore para el caso de $d = 1$ y $k = 1$ pero Knor ha demostrado que podemos encontrar digrafos radiales de Moore para cualquier valor de grado y radio.

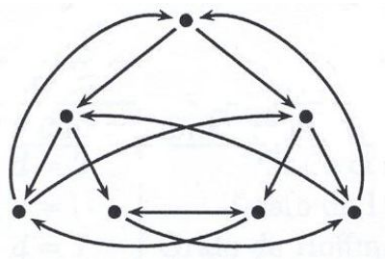

Figura 9: Ejemplo de digrafo radial de Moore de grado $d = 2$ y radio $r = 2$

Tanto los digrafos de Moore como los digrafos radiales de Moore deben ser fuertemente conexos y como se ha definido en el apartado anterior siguen la definición de excentricidad ofrecida por Knor. Además, la definición de los digrafos radiales de Moore también propuesta por Knor, enuncia que son $d - regulares$. Además de la regularidad, este tipo de estructuras no son simétricos, un vértice no puede tener arcos de ida y vuelta ya que en todos los niveles excepto en el último, el grado de salida de cada vértice ya está fijado, por lo que es imposible que regrese un arco.

En el apartado relacionado con las distancias y excentricidades se ha comentado que el primer paso para determinar si un digrafo es o no radial de Moore era el cálculo de la matriz de distancias mediante la utilización del algoritmo BFS. A continuación deberemos calcular la excentricidad de cada uno de los vértices, para ello, antes se han debido calcular las excentricidades tanto de entrada como de salida y a partir de allí la excentricidad de cada nodo siguiendo la definición de excentricidad de Knor. Cuando tenemos almacenada la excentricidad de cada uno de los vértices, se procederá a decir si el digrafo es o no radial de Moore y para ello se deben comprobar si las condiciones que se nombran en la definición de digrafo radial de Moore se cumplen o no, es decir, buscar el radio k y diámetro D entre las excentricidades resultantes y asegurarnos que $D \leq k + 1$.

4.3.1. Representación

El objetivo que tenemos con este proyecto es llegar a tener todos los posibles digrafos radiales de Moore, para ello debemos hacer una búsqueda de todos las posibles combinaciones de adyacencias entre una serie vértices, teniendo en cuenta que hay una estructura fijada de vértices y adyacencias. La idea del programa que tenemos que implementar es conseguir calcular dos

tipos específicos de digrafos radiales de Moore con un grado y un radio fijados inicialmente ya que el Mathematica no es capaz de calcularlos debido al gran coste de recursos y tiempo necesarios para resolverlo y otro ejemplo más sencillo que ya se ha calculado mediante otras técnicas para la comprobación de cálculos.

Como sabemos la definición y las propiedades de los digrafos radiales de Moore, ya podemos dar pie a explicar de forma general como se han representado las estructuras en el trabajo de programación.

Ya se ha comentado en el primer apartado que los digrafos se pueden representar de diferentes formas pero nosotros solamente nos fijaremos en la representación del digrafo mediante la matriz de adyacencia ya que se trata de la estructura más sencilla para la realización de nuestro cálculos. Durante el desarrollo del proyecto se han realizado una serie de modificaciones, por ejemplo, en un principio se utilizó la representación del digrafo mediante listas para realizar un conjunto de operaciones, pero esto lo comentaremos más adelante.

Para comenzar, hemos creado una clase llamada matriz_binaria que consta de una serie de operaciones básicas de matrices que nos serán imprescindibles para la creación de otra de las clases que se usarán, la clase digrafo, la cual utiliza la matriz de adyacencia para su representación.

También hay una clase lista con un grupo de operaciones básicas que permiten trabajar con determinada información de forma sencilla. Utilizando dicha clase se ha creado otra llamada digrafol que utilizará las listas para el almacenamiento de los digrafos mediante sus listas de adyacencia.

En un apartado posterior explicamos más detalladamente el funcionamiento y estructura de algunas de las clases, junto con sus operaciones. La clase digrafol no se comentará más profundamente ya que finalmente no se ha utilizado para la resolución del proyecto.

5. Estructura base

5.1. Introducción

En primer lugar sabemos que los digrafos que se deben calcular tienen una estructura subyacente fija durante toda la ejecución del programa. También conocemos el número de vértices que deben tener los grafos dirigidos, la profundidad máxima en la que deben estar situados los nodos, el grado de salida y de entrada de cada uno de los vértices del digrafo y un intervalo de valores que puede tener el diámetro del $D(G)$, que nos lo ofrece el enunciado en la definición de digrafos radiales de Moore. Lo que debemos averiguar son las combinaciones posibles de un gran número de arcos del nivel inferior que cumplan las propiedades de los digrafos radiales de Moore ya que el resto de la estructura estará fijada [9].

5.2. Características y representación

Como hemos dicho anteriormente, hay tres clases de digrafos radiales de Moore concretos y con unas características específicas sobre los que queremos operar y son los siguientes:

- Digrafo radial de Moore de grado $d = 3$ y radio $r = 2$.
- Digrafo radial de Moore de grado $d = 2$ y radio $r = 3$.
- Digrafo radial de Moore de grado $d = 2$ y radio $r = 2$. Este digrafo no es un objetivo directo del proyecto (ya que de ellos si se saben los resultados) pero lo utilizaremos para comprobar si el proyecto funciona correctamente ya que de este tipo específico de estructuras si que conocemos los digrafos resultantes que sean radiales de Moore y así, se podrán comparar con los que se obtengan con nuestro programa.

En primer lugar, se creará el digrafo G de tal forma que el vértice central 0 se encuentre situado en la parte superior, a distancia 0 de él mismo y con una distancia máxima k al resto de vértices. El grado de salida $g^+(0)$ será el correspondiente al del digrafo y el grado de entrada $g^-(0)$ será 0. El resto de vértices, excepto los que se encuentren en el nivel k, tendrán grado de salida igual al grado de salida del vértice central y grado de entrada igual a 1.

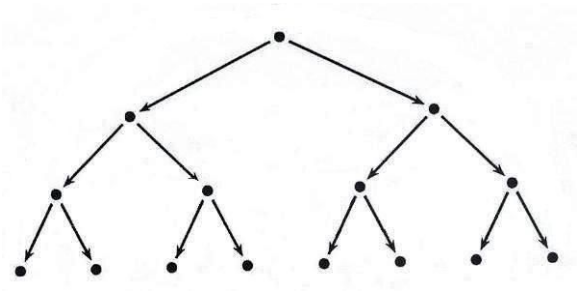

Figura 10: Representación de la estructura base de grado $d = 2$ y radio $r = 3$ sin las adyacencias fijadas

El siguiente paso será añadir una serie de arcos que serán fijos para todos los digrafos resultantes y que están explicados en el punto posterior. Con todo esto fijado quedarían por calcular todas las combinaciones posibles de arcos para conseguir que se cumplan las propiedades de los digrafos radiales de Moore y esto se conseguirá utilizando Backtracking y está explicado su funcionamiento en un apartado posterior.

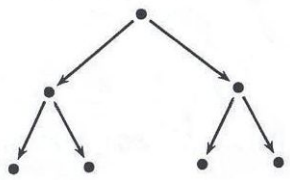

Figura 11: Representación de la estructura base de grado $d = 2$ y radio $r = 2$ sin las adyacencias fijadas

Habrá una serie de valores del grafo dirigido que serán guardados en atributos privados de la clase digrafo y serán fijos durante toda la ejecución del programa. A parte se creará la matriz de adyacencia junto con las uniones iniciales entre los vértices que no será fija sino que se irá modificando mientras el programa esté funcionando.

5.3. Vértices centrales

Después de crear la estructura base, hay unos arcos que van dirigidos desde el nivel k hacia un vértice central que se fijará siguiendo una proposición que aparece en [9]. Más adelante también tendremos una función que nos calcule el número de vértices centrales del digrafo en cuestión.

Definición: El centro de un digrafo G, denotado por $Z(G)$, es el conjunto de vértices con excentricidad igual al radio. A los elementos de $Z(G)$ se les llama vértices centrales.

Entonces, si λ denota un vértice central de un digrafo regular G de grado $d \geq 1$, radio $k \geq 1$ y orden $n_{d,k}$, para todo vértice $v \in N^+(\lambda)$ existe un único vértice $w \in N^-(\lambda)$ a distancia $k-1$ de v. El vértice w es único para cada v ya que sino existiría un $v' \in N^-(\lambda)$ cuya distancia a λ sería mayor que k y eso no es posible ya que $e^-(\lambda) = k$. Esta estructura estará fijada en todos los digrafos por lo enunciado anteriormente, pero no implica que en un digrafo solo pueda haber un vértice central.

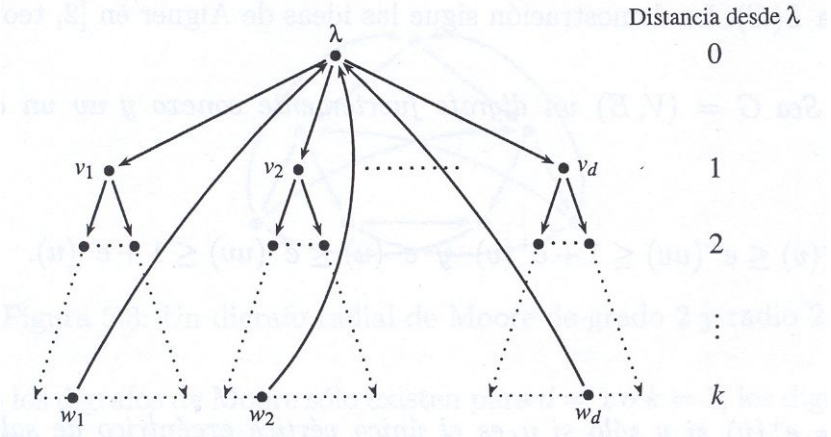

Figura 12: Estructura de un digrafo regular de grado d, radio k y orden $n_{d,k}$ 'colgado' desde un vértice central λ

En la figura podemos ver los arcos que se considerarán también como estructura fija y no variarán a la hora de crear los posibles digrafos radiales de Moore. Posteriormente, cuando ya tengamos el digrafo construido, uno de los cálculos que se realizarán será el número de vértices centrales del grafo dirigido que no nos es de utilidad para calcular si el digrafo resultante es radial de Moore sino para obtener información adicional del mismo.

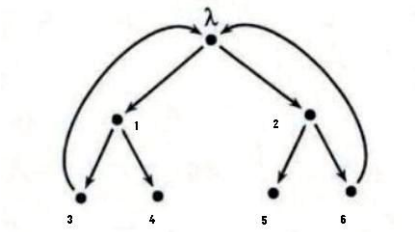

Figura 13: Ejemplo de una posible estructura base para digrafo de grado $d = 2$ y radio $r = 2$

6. Generación de digrafos

6.1. Introducción

Hay ocasiones en las que se debe resolver un determinado problema pero no existe un método específico para dar con su solución. Un ejemplo claro de esta situación se encuentra en el juego del ajedrez del cual no sabemos ningún método algorítmico que nos permita programar este juego siguiendo un esquema específico. Al no saberlo, lo que se hace es, a partir de la situación actual del tablero, explorar todas las situaciones a las que se llega después de la realización de todas las jugadas prometedoras y todas las situaciones a las que se llegará después de las jugadas que puede hacer el oponente como respuesta a cada una de nuestras posibles jugadas. Esta técnica solamente la utilizamos cuando no se sabe ningún esquema algorítmico que nos guíe a una solución.

En nuestro caso tenemos el mismo problema que en el juego del ajedrez ya que para la generación de los digrafos no hay ningún algoritmo concreto que sea bajo en coste, por lo que deberemos optar por la técnica de Backtracking que consiste en la exploración exhaustiva del espacio de búsqueda donde debemos encontrar la solución. Esta técnica siempre debe ser el último recurso a elegir a la hora de programar ya que el tiempo y coste de ejecución son elevados [12], [1], [6].

Antes de explicar detalladamente el Backtracking hay que comentar como se generan una serie de estructuras que almacenaran los datos esenciales a la hora de llamar a la función de Backtracking para que sea ejecutada.

6.2. Primera llamada a Backtracking

Cuando ya tenemos la estructura base se debe decidir como queremos almacenar las adyacencias entre vértices, cuantas veces se repetirán esos vértices cuando se deban buscar adyacencias, etc.

La forma en la que se van a guardar las adyacencias será en dos arrays de tamaño variable según el digrafo del que queramos calcular los valores, siendo el grado d, diámetro k y radio r:

◇ Array de vértices de salida: En todas las estructuras base de los digrafos vemos que el grado de salida es el que se especifica inicialmente en todos los niveles excepto en el k. Los vértices del nivel k van a tener $k-1$ o k arcos que salgan de ellos. Como hemos dicho anteriormente, hay un número determinado de vértices del nivel inferior que serán adyacentes con el vértice central λ, esos serán los que tengan $k-1$ arcos adyacentes con otros vértices de niveles superiores o del nivel k. El resto de esos vértices, es decir el número de vértices del nivel inferior menos d número de vértices, serán los que tengan grado de salida k.

-Array de vértices de salida de los digrafos de grado $d = 2$ y radio $r = 2$:

3	4	4	5	5	6

-Array de vértices de salida de los digrafos de grado $d = 2$ y radio $r = 3$:

7	8	8	9	9	10	10	11	12	12	13	13	14	14

-Array de vértices de salida de los digrafos de grado $d = 3$ y radio $r = 2$:

4	4	5	5	5	6	6	6	7	7	8	8	8	9	9	9	10	10	11	11	11	12	12	12

◇ Array de vértices de entrada: En la estructura base creada se puede observar que todos los vértices del nivel inferior son adyacentes con otros del mismo nivel o del nivel superior, eso implica que $g^-(v) = 1$ para $v \in V - \{\lambda\}$, esto es debido a que al existir adyacencias entre vértices del nivel k con λ, el vértice central ya tiene $g^-(v) = k$ y no debe haber ninguna adyacencia más entre cualquier vértice y λ.

En el array se almacenarán tantas repeticiones del número asociado al vértice como arcos incidentes necesite para obtener $g^-(v) = k$.

-Array de vértices de entrada de los digrafos de grado $d = 2$ y radio $r = 2$:

1	2	3	4	5	6

-Array de vértices de entrada de los digrafos de grado $d = 2$ y radio $r = 3$:

1	2	3	4	5	6	7	8	9	10	11	12	13	14

-Array de vértices de entrada de los digrafos de grado $d = 3$ y radio $r = 2$:

1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11	11	12	12

◇ Combinaciones de elementos del array de vértices de entrada: Este array contendrá todas las combinaciones que se realizarán en la función de Backtracking. Los elementos sobre los que se harán las combinaciones serán los elementos del array de vértices de entrada.

Esta función realizará la primera llamada a Backtracking, eso implica tener toda la estructura base del digrafo montada y todos los datos necesarios para calcular las posibles combinaciones de todos los digrafos.

6.3. Backtracking

En un principio, este método de búsqueda se situó en el círculo de la Inteligencia Artificial y se utilizó para resolver problemas para los que no se conocía ninguna técnica más eficiente por lo que se les denomina también métodos débiles para la resolución de problemas. La principal característica de esta técnica es que si no consigue encontrar una solución vuelve hacia atrás para probar otras posibles soluciones, de ahí proviene su nombre.

Se trata de una técnica general que consiste en recorrer de forma sistemática todos los posibles caminos. Cuando llegamos al punto en el que no hay ninguna solución, se retrocede al paso anterior en busca de otro posible resultado por otro camino distinto. Mediante este método sabemos que si existe una posible solución, seguro que se encuentra. Esta técnica también

se puede usar con la finalidad de encontrar todas las posibles soluciones de un espacio de búsqueda determinado, que es la opción en la que nos vamos a fijar nosotros.

El procedimiento general de Backtracking está basado en la descomposición del proceso de búsqueda en tareas parciales de tanteo de soluciones. Estas tareas parciales son recursivas y construyen gradualmente las soluciones.

El algoritmo de Backtracking esta basado en un recorrido hasta el fondo de un árbol, por lo que usa el método de búsqueda DFS (Depth First Search). Este árbol es conceptual e implícito y solo lo usaremos para entender cuales son los pasos que se siguen en el proceso de encontrar las soluciones.

Cada nodo del nivel k del árbol de posibles soluciones representa una posible solución y está formado por k etapas que ya han sido realizadas. Para examinar el conjunto de posibles soluciones es suficiente recorrer este árbol construyendo soluciones parciales mientras se avanza en el recorrido. Al final del recorrido nos podemos encontrar con varias situaciones:

- Que tenga éxito, o sea que se encuentre una solución. Si lo que buscábamos era una sola solución entonces el algoritmo termina aquí. Por el contrario, si lo que buscábamos eran todas las soluciones o la mejor de todas ellas, el algoritmo explorará el árbol buscando otras situaciones.
- Que no tenga éxito, en este caso se retrocede al nodo anterior y se buscan otros posibles caminos para alcanzar la solución.

Definición: Un algoritmo que utiliza backtracking para la resolución de un problema consiste, a partir de un inicio de solución (o solución parcial)S_p (pudiendo ser vacía inicialmente) a ese problema, intentar completarla de todas las formas posibles $(c_1, c_2, ..., c_n)$.

- Si una de estas formas $(S_p \bigcup c_i)$ da lugar a una solución completa S_c, el algoritmo termina con éxito o continúa buscando el resto de soluciones. Sino, se desestima y se prueba a completar S_p con la siguiente continuación posible $(S_p \bigcup c_{i+1})$. Este sería el paso de vuelta atrás.
- Si después de probar todas las posibilidades de completar S_p $(c_1, ..., c_n)$, ninguna de ellas permite encontrar una solución S_c, se llega a la conclusión que S_p no se puede completar hasta llegar a generar una solución completa.

Este es el algoritmo general que se ha aplicado al generar la función de Backtracking:

Algoritmo Backtracking

```
1:  tamano ⟵ (g(v) − 1)*(|V| − 1)
2:  V[tamano]
3:  si (k == tamano)
4:      guardar(C)
5:  fin si
6:  sino
7:      ncand ⟵ ConstruirCandidatos(k, V)
8:      para i = 0 hasta ncand − 1
9:          V[k] ⟵ V[i]
10:         Backtracking(k + 1)
11:     fin para
12: fin sino
```

6.4. Acotación de número de digrafos radiales de Moore

A continuación están explicadas las cotas máximas del número de cada clase de digrafos radiales de Moore que estamos estudiando. Para cada uno de los casos hemos realizado los cálculos en base a unas determinadas condiciones que no coinciden en todos los casos.

En digrafos radiales de Moore de grado $d = 2$ y radio $r = 2$:

Para obtener una cota en el número de digrafos radiales de Moore lo haremos por pasos. El primer paso va a ser calcular el número de secuencias de vértices sin que haya en ellas repetición alguna de elementos. En el segundo paso, incluiremos otra condición a la anterior que será que no importe el orden de los elementos entre los dos pares de posiciones con valores iguales, es decir, la segunda y tercera posición y la cuarta y la quinta. Se han repasado conceptos de la siguiente referencia bibliográfica [5].

Comenzaremos con la obtención del número de secuencias sin repetición de elementos:

$$6 \times 5 \times 4 \times 3 \times 2 \times 1 = 6! = 720$$

Las secuencias que se utilizan son las combinaciones de los elementos de los arrays de los vértices de entrada y se comparan con cada elemento de los arrays de salida. Pongamos un ejemplo:

3	4	4	5	5	6

2	3	5	6	4	1

El vector situado arriba es el array de vértices de salida de grado $d = 2$ y radio $r = 2$ y el que nos encontramos abajo, el array de vértices de entrada. Los arcos serán cada una de las adyacencias entre la posición i del array superior con el array inferior. En esta secuencia no nos encontramos con lazos, ya que ningún elemento de ninguna posición de los dos arrays coincide. Tampoco existe repetición de elementos, el array de vértices de entrada no tiene ningún elemento igual. Esto se aplica también en los digrafos radiales de Moore de grado $d = 2$ y radio $r = 3$ y grado $d = 3$ y radio $r = 2$.

El resultado escrito anteriormente se obtiene sabiendo que en la primera posición se puede poner 6 valores diferentes, en la siguiente, 5 solamente ya que un elemento ha sido ya situado en la posición inicial. En cada posición posterior tendremos como candidatos la diferencia en uno de la posición anterior y así sucesivamente hasta llegar a la posición 6.

Ahora obtendremos el número de secuencias sin que importe el orden entre las posiciones 2^a y 3^a y 4^a y 5^a junto con la condición anterior:

$$\binom{6}{2} \times \binom{4}{2} \times 2 = 180$$

Este resultado se obtiene desglosando el problema. Primero hay que fijarse en los cuatro elementos (repetidos dos a dos) y los valores que pueden tomar. Inicialmente hay 6 elementos que son los que puede tomar uno de los valores de una de las parejas. El otro, solamente tiene 5 elementos para ser insertados. De este resultado solo nos interesan la mitad de valores ya que, como se ha dicho, no importa el orden entre los elementos de la pareja. En la otra pareja tenemos 4 elementos (ya que 2 han sido introducidos anteriormente) para el primer valor y 3 para el otro. En la penúltima posición solo tendremos 2 elementos para introducir y en la última 1 solamente.

En digrafos radiales de Moore de grado $d = 3$ y radio $r = 2$:

En este caso miraremos solamente las secuencias que obtendremos que cumplan la primera condición del caso anterior ya que los cálculos teniendo en cuenta que no importe el orden entre posiciones con elementos repetidos se hace más complicada. Por lo tanto, el resultado será:

$$\binom{24}{2} \times \binom{22}{2} \times \binom{20}{2} \times \ldots \times \binom{2}{2} = \frac{24!}{(2!)^{12}} \simeq 151 \times 10^{18}$$

La idea que se debe tener en cuenta en este caso es la obtención de combinaciones en m espacios diferentes de n elementos determinados. Esto se realiza siempre de la misma forma y se calcula de la siguiente manera: $\binom{m}{n}$.

En digrafos radiales de Moore de grado $d = 2$ y radio $r = 3$:

En este caso también vamos a calcular el número de secuencias que cumplan las dos condiciones. Comenzaremos con la obtención del número de secuencias sin repetición de elementos:

$$14! \simeq 871 \times 10^{8}$$

Se sigue la misma deducción que en la obtención de digrafos radiales de Moore con grado $d = 2$ y radio $r = 2$.

Ahora obtendremos el número de secuencias (sin que importe el orden) entre parejas de elementos repetidos, que en este caso tenemos 6 parejas de elementos repetidos que se deben tener en cuenta a la hora de realizar las combinaciones.

$$\binom{14}{2} \times \binom{12}{2} \times \binom{10}{2} \times \binom{8}{2} \times \binom{6}{2} \times \binom{4}{2} \times 2 \simeq 136 \times 10^{7}$$

6.5. Construir los posibles candidatos

El Backtracking, a parte de la función principal, está formado por otras dos funciones muy importantes para su ejecución. Una de ellas es aquella que vemos en el algoritmo anterior que es la de ConstruirCandidatos cuyo objetivo es ser llamada para construir los posibles candidatos durante la ejecución del Backtracking. La otra función es más trivial y se encarga de realizar la acción oportuna con el resultado obtenido en cada iteración que ha variado durante el desarrollo del proyecto.

Algoritmo ConstruirCandidatos

```
1:  numcand ⟵ 0
2:  tamano ⟵ (g(v)−1) ∗ (|V| − 1)
3:  contador ⟵ 0
4:  para s = 0 hasta s < tamano
5:      ¬repetido
6:      ¬formanlazo
7:      sirve
8:      para j = 0 hasta j < k
9:          si (E[s] == C[j])
10:             si (contador < (grado − 2))
11:                 repetido
12:                 contador ⟵ 0
13:             fin si
14:         fin si
15:     fin para
16:     si (S[k] == E[k])
17:         formanlazos
18:     fin si
19:     si (k == 1)
20:         si (E[s] == C[k − 1]) y (S[k − 1] == S[k])
21:             ¬sirve
22:         fin si
23:     fin si
24:     si (k > 1)
25:         si ((E[s] == C[k − 1]) y (S[k − 1] == S[k])) o
26:            ((E[s] == C[k − 2]) y (S[k − 2] == S[k]))
27:             ¬sirve
28:         fin si
29:     fin si
30:     si (¬repetido y ¬formanlazos y sirve)
31:         V[numcand] ⟵ E[s]
32:         numcand ⟵ numcand + 1
33:     fin si
34: fin para
35: return numcand
```

Para entender su funcionamiento vamos a explicar la función Construir-Candidatos. En nuestro caso hay una serie de puntos que hay que tener en

cuenta.

La función ConstruirCandidatos es la base para obtener las combinaciones resultantes de adyacencias entre los vertices V del digrafo $G = (V, A)$. Aquí se acota el número de resultados de combinaciones proponiendo tres condiciones a la hora de la creación de las mismas. Estas son:

- Que no haya lazos en la creación de arcos A entre los vértices del digrafo ya que, como hemos explicado anteriormente, no tiene sentido debido a que una de las principales características a tener en cuenta a la hora de la representación de casos reales utilizando digrafos es minimizar la distancia entre los nodos del digrafo y los lazos no resultan de utilidad en este punto.

- Que no haya elementos repetidos en la combinación resultante. En los casos de grado $d = 2$ y radio $r = 2$ y en el de grado $d = 2$ y radio $r = 3$ no hay ningún problema, solo debemos tener en cuenta que no haya ninguna repetición entre elementos. El principal inconveniente viene en el caso de grado $d = 3$ y radio $r = 2$ que nos encontramos con una serie de repeticiones ya que en la mayoría de vértices entra más de un arco, eso implica tener un control sobre el número de vértices utilizados en las combinaciones.

  En la programación, esto se consigue de la siguiente forma:

  - Para tratar los dos primeros casos nombrados anteriormente, debemos comparar dos posiciones de dos arrays y comprobar que no coincidan.
  - En el otro caso debemos tener un contador de número de veces que se repite un determinado valor, a parte también de la condición anterior.

  Las dos condiciones se comprueban en los tres casos pero la segunda opción solamente se ejecutará para el caso de grado $d = 3$ y radio $r = 2$.

- Que no tengamos dos o más vértices iguales adyacentes a un mismo vértice ya que se obtendrían arcos múltiples.

  Miramos de incluir otra condición en esta función para hacer más eficiente el programa pero nos resultó imposible ya que no encontramos como hacerlo. El objetivo que tenía era no guardar las combinaciones iguales. Para ello debíamos mirar que al comparar dos combinaciones, los elementos repetidos del array de vértices de salida no tuvieran asociados elementos iguales del array de vértices de entrada.

  Después intentamos realizar eso mismo en otra función, cuando tuviéramos todas las combinaciones posibles. Para el caso de grado $d = 2$

y radio $r = 2$ era relativamente fácil y eficiente pero para los dos otros casos era muy complicado e ineficiente.

6.5.1. Número de posibles resultados repetidos

Como hemos dicho anteriormente, la idea que se tuvo en un principio fue comprobar si existían digrafos repetidos antes de guardarlos en documentos de texto. Inicialmente se quiso incluir una condición en la función Construir-Candidatos pero no se encontró la forma. Luego se intentó crear una función, que después de almacenar las secuencias resultantes, eliminara de la lista los digrafos iguales pero también nos encontramos con problemas de crecimiento exponencial de valores. En este apartado vamos a mostrar como de grandes se hacían esos valores, lo que nos servirá como demostración de porque no se puede realizar esa función. Se van a explicar por separado cada uno de los casos para un mejor entendimiento.

La no aparición de digrafos repetidos implica que el orden de las combinaciones asociadas a varios elementos iguales no importa.

- Digrafos radiales de Moore de grado $d = 2$ y radio $r = 2$: En este caso, el número de valores iguales no es muy elevado ya que no se pueden obtener muchos digrafos repetidos debido a que solamente tenemos dos parejas de dos elementos que pueden contener valores repetidos. El resultado será

  $$(2!)^2 = 4$$

  siendo 2! las posibles combinaciones de dos elementos. El significado de este resultado significa nos encontramos con 4 secuencias iguales de un mismo digrafo. Eso equivale a realizar pocas comprobaciones ya que en este caso se obtienen 144 secuencias y al realizar el recorrido se eliminan las secuencias iguales, por lo que se reduce el número de comprobaciones.

- Digrafos radiales de Moore de grado $d = 2$ y radio $r = 3$: En este caso tenemos 6 parejas de 2 elementos lo que nos resulta el valor siguiente:

  $$(2!)^6 = 64$$

  siguiendo la misma lógica que en el caso anterior, por lo que tendremos 64 digrafos iguales de un mismo digrafo. Como podemos ver, en este caso, la comprobación de digrfos iguales sería muy costosa.

- Digrafos radiales de Moore de grado $d = 3$ y radio $r = 2$: En este caso se dispara el resultado ya que tenemos grupos de 2 elementos y de 3. Hay 3 grupos de 2 elementos en cada uno y 6 de 3 elementos. El resultado será el siguiente, siguiendo también las mismas deducciones que en el apartado anterior:

$$(2!)^3 \times (3!)^6 = 373248$$

  En este caso sería muy complicado y costoso realizar la comprobación debido a que nos encontramos 373248 digrafos iguales de un mismo digrafo.

7. Isomorfismo de los digrafos

7.1. Introducción

Los resultados que se habrán obtenido realizando lo explicado anteriormente serán un conjunto de digrafos donde no hay distinciones en si son o no isomorfos. Durante la realización del proyecto se han intentado implementar una serie de herramientas que han fracasado debido a unos problemas que explicaremos posteriormente.

7.2. Isomorfismo

Definición: Diremos que dos digrafos $G = (V, A)$ y $G' = (V', A')$ son isomorfos, y lo denotaremos por $G \simeq G'$, si existe una aplicación biyectiva f de V en V' tal que preserva las adyacencias, es decir,

$$\forall u, v \in V,\ u \sim v \Leftrightarrow f(u) \sim f(v).$$

Podemos decir, de una forma intuitiva, que dos digrafos isomorfos solamente se diferencian por la forma en que se etiquetan sus vértices y, en general, por su representación gráfica. Los vértices de dos digrafos isomorfos se pueden identificar uno a uno de tal manera que las adyacencias de uno y otro sean las mismas, es decir, la estructura de los dos digrafos, una vez identificados los vértices correspondientes, es idéntica. En particular, dos digrafos isomorfos tienen el mismo número de vértices y el mismo número de aristas.

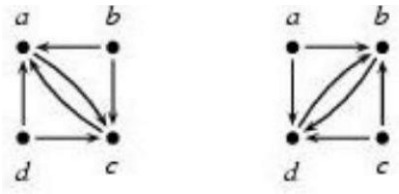

Figura 14: Representación gráfica de dos digrafos isomorfos

7.3. Cálculo del isomorfismo

Una de las funciones que hubiera tenido que tener este proyecto es que eliminara los digrafos radiales de Moore isomorfos pero se han dado una serie de problemas en su ejecución. Se han utilizado dos técnicas en el momento de su cálculo que serán las que explicaremos a continuación detalladamente.

8. Cálculo del isomorfismo empleando certificados

8.1. Introducción

Para el cálculo de isomorfismos se han probado dos técnicas las cuales han fracasado debido a una serie de problemas que nos han generado. Igualmente se ha creído conveniente su explicación porque se trata de formas interesantes del cálculo del isomorfismo para grafos dirigidos.

En este punto no nos centraremos demasiado, debido a que la herramienta que se utiliza ha sido creada por otra persona y nosotros solamente la utilizaremos, sin reparar en su funcionamiento, en comprobar si podemos utilizarla correctamente en nuestro proyecto. Si se desea conocer la explicación del procedimiento para el cálculo del isomorfismo y la explicación detallada sobre el mismo, consultar el TFC [7].

8.2. Problemas

El programa no dio ningún error de compilación, se agregaron una serie de operaciones necesarias a la clase digrafo y seguidamente se optó por probar su funcionamiento. El resultado que obtuvimos fue que la herramienta funcionaba solamente con unos digrafos con unas características específicas, y algunos de nuestros digrafos no las cumplían. Debido a este problema, decidimos cambiar de técnica para intentar calcular el isomorfismo. El apartado siguiente explicará detalladamente todo aquello que se ha utilizado de dicha técnica y su funcionamiento.

9. NAUTY

9.1. Introducción

Una de las ideas en el momento de implementar nuestro programa era que realizara el testeo de isomorfismo entre digrafos pero el resultado no ha sido el esperado. La segunda opción que hemos utilizado para calcular los isomorfismos es mediante el Nauty que nos permite representar los digrafos y usando un conjunto de herramientas denominadas *gtools*, determinar el isomorfismo [11].

La primera pregunta que nos hacemos es: ¿Que es NAUTY? Pues NAUTY es una herramienta que nos permite representar grafos tanto dirigidos como no dirigidos y realizar un conjunto de operaciones sobre ellos, además de obtener información sobre los mismos de forma sencilla y rápida. También, dentro de Nauty, hay otra herramienta que tendremos que instalar para poder realizar una serie de acciones sobre los digrafos y que se denomina gtools.

Inicialmente, para utilizar esta herramienta se deben instalar un conjunto de paquetes que contienen los utensilios necesarios para realizar las operaciones que nos interesan. A continuación, para poder utilizar Nauty solo es necesario incluir en el programa que se desean realizar las operaciones, la librería nauty.h, que incluye los procedimientos y los tipos básicos para crear grafos y realizar operaciones elementales sobre los mismos, y gtools.h si se quieren utilizar las operaciones que contiene para aplicar a los digrafos.

Nauty es una herramienta que ha sido creada por Brendan D. McKay que nos permite realizar un conjunto muy variado de operaciones sobre grafos y digrafos.

En este apartado profundizaremos un poco en esta herramienta y explicaremos los problemas con los que nos hemos encontrado durante la utilización de la misma.

9.2. Instalación y uso

El autor de esta herramienta tiene un sitio web donde se puede preguntar cualquier cuestión relacionada con la misma, esto implica la mejora de la misma mediante las aportaciones y comentarios de los usuarios que ayudan a una mejora continua de Nauty. Por ello debemos tener en cuenta en el momento de instalar la herramienta que vamos a utilizar la versión más

actualizada. Esto lo podemos encontrar en el README que está incluido en el conjunto de ficheros comprimidos y que se descarga de la página de McKay.

La aplicación Nauty está formada por un conjunto de ficheros sobre los cuales no profundizaremos ya que no nos es de interés conocer la estructura de dicho programa sino su funcionamiento.

Seguidamente se explicará la instalación y configuración de la herramienta para poder comenzar con su utilización. Antes de comenzar con la instalación, se deben extraer los archivos comprimidos que nos hemos bajado de la página correspondiente.

En este proyecto, se va a instalar la herramienta en un Linux Ubuntu, por lo que realizaremos la instalación como explicaremos a continuación. En primer lugar, antes de realizar nada, debemos estar situados mediante la utilización de un terminal en la carpeta donde se han extraído los archivos. Allí escribiremos ./configure para crear los ficheros que vamos a utilizar para la representación de los digrafos utilizando el lenguaje del Nauty. Junto a este conjunto de archivos, también se creará el Makefile que será el encargado de compilar Nauty realizando la llamada make nauty. Además debemos compilar también gtools que es una librería que ofrece una serie de herramientas para trabajar con grafos tanto dirigidos como no dirigidos.

Finalizados estos pasos, ya podemos comenzar a usar la herramienta. Para ello solamente debemos incluir en nuestro programa el fichero "$nauty.h$" y "$gtools.h$", además de declarar las variables relevantes. Las llamadas a las funciones que necesitemos las realizaremos en nuestro programa.

9.3. Operaciones

En nuestro proyecto solo nos fijaremos en una serie de operaciones específicas y en declarar unas variables determinadas para llamar a esas funciones.

Comenzaremos enumerando las variables que vamos a declarar y que significado tienen cada una de ellas:

- $graphg[MAX * MAX]$ es el grafo en formato de nauty sobre el cual realizaremos el test de isomorfismo. Un grafo viene representado por un array de n sets. El i-set representa los vértices a los cuales i es adyacente.

Un set está representado por un array de m enteros representados por 16, 32 0 64 bits dependiendo del compilador. Esos bits están numerados $0, 1, ..., n$ de izquierda a derecha. los bits que no tienen valor se les considera que tienen valor 0.

- *amtog* es la función que lee un digrafo en formato de matriz de adyacencia y lo pasa al formato que utiliza el nauty. La forma que tiene la llamada a amtog es la siguiente:

  $amtog[-n\#sghq][infile[outfile]]$ Lee los grafos en formato de matriz de adyacencia

  $-n\#$ Establece el orden del grafo inicial a # (no predeterminado). Puede ser sobrescrito en el grafo de entrada.

  $-g$ Escribe el grafo de salida en formato graph6(predeterminado).

  $-s$ Escribe el grafo de salida en formato sparse6.

  $-h$ Escribe una cabecera (de acuerdo con $-g$ o $-s$).

  $-q$ Suprime la información auxiliar.

  El grafo de entrada trata de una secuencia de comandos restringida a:

  $n = \#$ Establece el número de vértices (no predeterminado) El = es opcional.

  m Matriz a seguir (01 con o sin espacio). m se asume también si 0 o 1 está contabilizado.

  M Es la complementaria de la matriz a seguir.

  t El triángulo superior de la matriz a seguir, línea a línea excluyendo la diagonal. (01 con o sin espacio)

  q salir (opcional)

  En nuestro caso no nos interesa casi ninguna de las opciones anteriores excepto $-g$ y $-s$. El resto no sabemos de forma específica que realizan pero no tiene importancia ya que no se usarán.

- *options* son las opciones que se le definen a la función. En nuestro caso, como vamos a trabajar con digrafos, se ha de igualar el booleano $options.digraph = TRUE$ y así le indicamos a la función que llamaremos posteriormente que le pasamos un digrafo.

- Parámetro *canong* hace que nos devuelva o no el etiquetado canónico del digrafo. El etiquetado canónico de un digrafo es el certificado de isomorfismo, es decir, el etiquetado canónico de dos digrafos G_1 y G_2

coinciden si y sólo si dichos digrafos son isomorfos. Si queremos saber si los dos digrafos son isomorfos, debemos activar el parámetro canong.

- función *nauty*() que recibe por parámetro una serie de valores. Los que debemos tener en cuenta son: g, *options*, *canong*. Se le pasan más parámetros pero a nosotros no nos interesa saber su funcionamiento, los declararemos de forma predeterminada como vemos en los ejemplos [11]. Esta función nos dará el etiquetado canónico del digrafo que le pasemos por parámetro. Para saber si dos digrafos son o no isomorfos, debemos comparar los resultados de las dos llamadas a *nauty*() (cada una con un digrafo diferente).

9.4. Problemas

Como hemos dicho, inicialmente se quería comprobar el isomorfismo de los digrafos radiales de Moore resultantes y así guardarlos para poder utilizarlos posteriormente.

Consideramos tratar este objetivo mediante el uso del Nauty pero operar con digrafos nos daba una serie de problemas a la hora de llamadas a funciones y declaración de variables. Estuvimos indagando y no se consiguieron solucionar estos problemas. Los ejemplos que se encontraron que trabajaban con Nauty eran programas realizados en C, por lo que, tal vez, el problema se debía a la utilización de Programación Orientada a Objetos, pero no se aseguró.

9.5. Solución alternativa

Finalmente, se realizaron los cálculos sobre los resultados obtenidos mediante el Mathematica, que al ser pocos no tardaron demasiado en finalizar. Si el programa hubiera finalizado su ejecución, dando como resultado todos los digrafos radiales de Moore posibles, no hubiera podido calcularse el isomorfismo ya que el número de digrafos sería muy elevado en los casos de grado $d = 2$ y radio $r = 3$ y grado $d = 3$ y radio $r = 2$.

No se trata de un método muy eficaz si nos encontramos con una gran cantidad de digrafos pero nos sirvió para comprobar si funciona el programa para grado $d = 2$ y radio $r = 2$.

10. Guardar las combinaciones

Durante el desarrollo del proyecto llegamos al punto de decidir como se guardan los digrafos para operar sobre ellos posteriormente. Se han probado varias opciones, las cuales tenían una serie de problemas que enunciaremos en los puntos siguientes.

Un inciso antes de explicar las versiones es comentar donde se crea la función o funciones que guarden, en un formato legible por el Mathematica, los digrafos resultantes. Decidimos hacerlo en el programa principal en vez de en la clase digrafo debido a que nos resultaba mas sencillo llamar a la función con el formato con el que estábamos trabajando en el momento de implementar la función. Además, también hemos tenido cambios en la forma de almacenar los digrafos. Inicialmente se querían guardar todos los digrafos en un mismo archivo pero finalmente se decidió guardar un digrafo en cada archivo por comodidad a la hora de operar posteriormente.

El formato en el que se guardarán los digrafos será mediante listas de adyacencia ya que se trata de una manera de representación que es compatible con el Mathematica.

10.1. Versión 1

Para guardar los digrafos resultantes se han probado varias opciones con sus pros y sus contras. El primer punto a tener en cuenta es si se comprueba si una estructura es radial de Moore o no justo después de cada combinación o si se guardan todas las combinaciones y finalmente se comprueban una a una.

Inicialmente decidimos si un digrafo es o no radial de Moore después de obtener cada resultado y seguidamente guardar en una estructura creada en memoria cada uno de los resultados que dieron positivo en esa comprobación. Luego guardaríamos cada uno de los resultados en un archivo de texto para operar sobre el.

En primer lugar, la estructura que se consideró utilizar fue una Lista donde almacenaríamos un digrafo formado por matriz de adyacencia o por lista de adyacencias. Al tener problemas con el almacenamiento del digrafo representado mediante lista de adyacencias, optamos por guardar el digrafo representado mediante matriz. Seguidamente optamos por almacenar en dicha Lista, un array que contendría los vértices de entrada. El cambio se hizo debido a problemas de espacio en memoria que comentaremos a continuación.

Problemas:

Para un digrafo pequeño, la estructura de tipo Lista que se crearía en memoria sería mínimo pero en el caso de grado $d = 2$ y radio $r = 3$ y el de grado $d = 3$ y radio $r = 2$, las estructuras que se crearían serían enormes y la memoria de la máquina sería insuficiente para poder crear todos y cada uno de los digrafos radiales de Moore.

Seguidamente está el dilema de escoger que elementos se guardarían en dicha lista. Se probó por almacenar en un principio digrafos representados mediante matrices de adyacencia pero seguidamente optamos por guardar listas debido a que el espacio en memoria que ocuparían sería algo menor y ya que se tendrían que guardar muchos digrafos, ampliaríamos la capacidad de almacenamiento de la memoria.

10.2. Versión 2

Cuando probamos la primera opción nos dimos cuenta de que el ordenador era capaz de resolver solamente 591 digrafos radiales de Moore, lo que suponíamos que era debido a la capacidad de almacenamiento de la memoria y decidimos guardar directamente, después de comprobar si un digrafo era radial de Moore o no, en un fichero de texto para no saturar la memoria.

Problemas:

Seguíamos teniendo problemas con la saturación de memoria y se pensó que sería por incluir en el Backtracking las funciones necesarias para la determinación de si el digrafo era o no radial de Moore.

10.3. Versión 3

Debido al problema enunciado en la Versión 2, decidimos guardar todas las combinaciones en archivos de texto y al finalizar, realizar el testeo de si el digrafo es o no radial de Moore, sobre cada una de las combinaciones obtenidas.

Problemas:

La saturación de memoria al ejecutar las funciones seguía e investigamos su procedencia y llegamos a la conclusión de que se producía en el momento

de calcular la matriz de distancias. Llegados a este punto, no conseguimos descubrir ninguna forma de calcular una mayor cantidad de digrafos radiales de Moore así que nos quedamos en este punto.

10.3.1. Conclusión

Al ejecutar el programa, en el caso de grado $d = 2$ y radio $r = 2$ el programa terminaba correctamente ofreciendo un conjunto de digrafos radiales de Moore, a los que se les pasó un test de isomorfismo por el Mathematica y cuyo número de resultados era el correcto.

Respecto a los otros casos, en el caso de grado $d = 3$ y radio $r = 2$ no se probó el programa ya que su ejecución sería más costosa que en el caso de grado $d = 2$ y radio $r = 3$. En este último caso llegamos a obtener 591 digrafos radiales de Moore.

11. Implementación del programa

11.1. Introducción

Este punto contendrá una explicación respecto a la parte de programación: lenguaje utilizado para su desarrollo, características sobre ese lenguaje, definición y explicación de las clases y profundización en algún punto de las mismas,...

11.2. Lenguaje utilizado

Para el desarrollo del proyecto se ha utilizado el lenguaje de programación C++. En un principio surgió la duda de si elegir C o C++ pero nos decantamos por C++ ya que su posibilidad de implementación mediante el uso de la Programación Orientada a Objetos nos hacía la programación más sencilla y estructurada, a parte de ofrecernos la posibilidad de la reutilización de código y capacidad de abstracción.

11.2.1. Características

Vamos a enunciar un conjunto de características que tiene el C++ y por las cuales lo hemos elegido para realizar el programa [13]:

- Los programas realizados con C++ se caracterizan por ser compactos y rápidos en su ejecución. Eso implica que su interfaz no esté muy elaborada y sea vista en modo texto. En nuestro caso, no deseamos que el programa sea "*bonito*" sino que sea rápido por lo que este lenguaje lo cumple.

- Es transportable, o sea, puede ejecutarse desde cualquier máquina y sistema operativo. Sería adecuado poder utilizarse para varios sistemas operativos ya que el Mathematica también lo hace.

- Es un lenguaje con muchos años de vida, casi 20, y parece que su vida se alargará ya que su uso no se debilita demasiado.

- Se trata de un Lenguaje de Programación Orientado a Objetos y como hemos comentado, es un lenguaje que consigue estructurar de forma adecuada el programa, reutilizar código, si es necesario y tiene un nivel de abstracción necesario para programar nuestro código.

11.3. Desarrollo de la aplicación

11.3.1. Clase Lista

Esta clase define la estructura de las listas y se trata de una lista genérica, es decir, es capaz de almacenar cualquier tipo de dato: entero, digrafo, carácter... Esta clase es hija de la clase estructura de datos (ed) y hereda de su padre un conjunto de métodos y atributos comunes a todas las estructuras de datos.

La clase en sí no ha sido creada íntegramente, sino se ha aprovechado parte del código de un TFC [7] y de prácticas realizadas en una asignatura de la carrera y se ha mejorado según hemos creído oportuno para que las operaciones se adaptaran a la finalidad de nuestro proyecto.

En una clase es necesario que existan unas operaciones básicas: consultoras, constructoras y creadoras. En ésta se encuentran varias de cada tipo para conseguir una rápida y fácil creación de la estructura, un fácil acceso a los datos almacenados en la estructura, una sencilla inserción de los mismos y un borrado rápido de la estructura junto con sus datos.

Hay tres operaciones encargadas de insertar elementos en una lista: una de ellas añade un elemento al final de la misma, otra al principio y la otra en la última en la posición apuntada por el iterador.

Nos encontramos con otras tres que eliminan elementos de una lista y otras tres que nos muestran elementos de la misma. Los elementos a eliminar o consultar también se realizan del final, inicio o del elemento apuntado por el iterador, como se ha enunciado en las operaciones de inserción.

También hay dos operaciones que nos dicen si se encuentra un determinado elemento en la lista. Una de ellas es más específica y busca el elemento en el nodo de la lista apuntado por el iterador. El resto de operaciones no es necesario comentarlas ya que su funcionamiento es muy sencillo.

A continuación están las cabeceras de las operaciones de la clase junto con sus atributos privados:

```
class Lista:public ED {
    nodo<T>* head;
    nodo<T>* tail;
```

```
public:
    friend class IteradorLista<T>;
    class noMem;
    Lista();
    Lista(Lista<T>&);
    ~Lista();
    void operator=(Lista<T>&);
    void insertarInicio(T&);
    void insertarFinal(T&);
    void insertar(T&, IteradorLista<T>&);
    void eliminarInicio();
    void eliminarFinal();
    void eliminar(IteradorLista<T>&);
    void consultarInicio(T&);
    void consultarFinal(T&);
    void consultar(T&, IteradorLista<T>&);
    void sustituir(IteradorLista<T>&, T);
    int numeroElementos();
    void vaciar_lista();
    bool Esta(T&);
    bool Esta(T&, IteradorLista<T>&);
    void buscar(T&, IteradorLista<T>&);
    bool vacia();
};
```

Esta es la clase utilizada para crear los nodos que se encuentran en la clase lista. Solo contiene atributos privados que los definen y su creación es necesaria para utilizar el tipo de lista que hemos creado: lista encadenada.

```
class nodo {
public:
    nodo<T>* sig;
    nodo<T>* ant;
    T valor;
};
```

Ya que la clase Lista se trata de una clase más especifica que una estructura de datos, se ha reescrito el código de una operación de la clase estructura de datos. Esto se ha realizado porque en esta clase, esta función se comporta de manera diferente. Se trata de la función bool vacia().

Además se ha implementado una función privada, o sea, que no puede utilizar el usuario (solamente puede ser usada por los métodos de la clase), para minimizar el tamaño del código haciéndolo más elegante.

11.3.2. Clase IteradorLista

En el punto anterior vemos que hay un conjunto de operaciones a las que les pasa por parámetro un IteradorLista ya que es necesario para su funcionamiento. Un iterador en una lista sirve para controlar la posición en una lista por lo que hay un conjunto de operaciones que se tienen que vigilar cuando se crea un iterador asociado a una lista determinada. La clase ha sido creada para que el iterador esté relacionado con listas genéricas. Esto se especifica en su clase padre. Una lista puede o no, tener asociado un iterador ya que si la lista no realiza operaciones que utilicen el iterador no es necesaria su utilización.

```
class IteradorLista:public Iterador<T>
    Lista<T>* ref;
    nodo<T>* ed;
public:
    friend class Lista<T>;
    IteradorLista(Lista<T>& l);
    void situarInicio();
    void situarFinal();
    IteradorLista<T>& operator++();
    IteradorLista<T>& operator++(int i);
    IteradorLista<T>& operator–();
    IteradorLista<T>& operator–(int i);
    void operator=(IteradorLista<T>& it);
    IteradorLista<T>& IteradorLista<T>::operator++(int i);
    IteradorLista<T>& IteradorLista<T>::operator−−(int i);
    bool inicio();
    bool fin();
    bool operator ==(IteradorLista<T>& it1,IteradorLista<T>& it2);
}
```

La clase IteradorLista no se trata de una estructura de datos sino de una herramienta que facilita operar sobre determinadas estructuras y es hija de una clase denominada Iterador que contiene las mismas operaciones que la clase IteradorLista pero sin realizar su implementación e igualadas a cero. A esas operaciones se las denomina virtuales puras y se crean con la finalidad

de ser implementadas en las clases más específicas, en este caso en la clase IteradorLista. Si se hubiera dado el caso de crear un iterador asociado a otra clase lista implementada de diferente forma que la que tenemos, se hubiera creado otra clase denominada de manera diferente y con las mismas operaciones de la clase Iterador pero con distinta implementación a la de la clase IteradorLista.

11.3.3. Clase Digrafo

Nuestra aplicación utiliza como estructura de datos principal el digrafo. Esta clase es la encargada de ofrecernos las operaciones necesarias para poder desarrollar el programa principal. Sus atributos son el orden, la medida y la matriz de adyacencias ya que la representación del grafo dirigido en esta clase no es mediante su lista de adyacencias sino por su matriz de adyacencias.

```
class digrafo {
    matrizbinaria* adyacencias;
    int orden;
    int medida;
public:
    class no_mem;
    digrafo();
    digrafo(int ord);
    digrafo(digrafol* d);
    digrafo(digrafo* d);
    ∼digrafo();
    int medidad();
    int ordend();
    int grado_salida(int vertice);
    int grado_entrada(int vertice);
    bool adyacentes(int v1, int v2);
    void conectar(int v1, int v2);
    void desconectar(int v1, int v2);
    void llenar_restantes(int& i, Lista<int>* restantes, IteradorLista<int>*
ite);
    void calcular_distancias(int& inicial, int* distancia);
    void calcular_excentricidades(matriz* m);
    void obtener_secuencia_excentricidades_salida(matriz* excentricidades, int*
secuencia);
    void obtener_secuencia_excentricidades_entrada(matriz* excentricidades,
int* secuencia2);
```

```
    void obtener_secuencia_excentricidades(int* secuencia, int* secuencia2, int*
secuencia3);
    int calcular_radio(int* secuencia);
    int calcular_diametro(int* secuencia);
    int calcular_vertices_centrales(int radio, int* secuencia3);
    bool Es_Radial_Moore(int& dia, int& rad, int& radio);
    digrafo& operator =(digrafo& d);
    bool operator ==(digrafo& d);
    bool operator !=(digrafo& d);
    friend ostream& operator<<(ostream& os, digrafo& d);
}
```

Hay varias funciones a destacar en esta clase y de las que es oportuno describir de forma poco detallada su funcionamiento ya que en apartados anteriores se ha profundizado sobre su desarrollo.

La operación void calcular_excentricidades() es la que se encarga de calcular la matriz de distancias del digrafo, de tamaño $n \times n$, y para ello utiliza void calcular_distancias() que se encarga de calcular la distancia de un vértice al resto utilizando el algoritmo BFS. Esta función utiliza otra muy sencilla, que no es interesante explicar, para alcanzar su objetivo.

La operacion void obtener_secuencia_excentricidades_salida() calcula la secuencia de excentricidades de salida, es decir, de cada fila de la matriz de distancias guarda el mayor valor, o lo que es equivalente, la mayor distancia entre dos vértices. La operación void obtener_secuencia_excentricidades_entrada, calcula la secuencia de excentricidades de entrada, es decir, de cada columna de la matriz de distancias guarda el mayor valor, o lo que es equivalente, la mayor distancia entre dos vértices.

11.3.4. Clase MatrizBinaria

Esta clase define una estructura necesaria para la representación de los digrafos. La clase matriz binaria ofrece un conjunto de operaciones que nos permiten crear la matriz de adyacencia del digrafo. Está formada como cualquier matriz: por filas, por columnas y por una estructura matricial donde almacenar valores enteros. Esos valores serán 0 y 1 ya que se guardaran todas las adyacencias entre los vértices del grafo dirigido.

Esta estructura no se utiliza en el programa principal, solamente nos sirve de ayuda a la hora de representar los digrafos. Las operaciones que nos ofrece

esta clase son las básicas para crear, eliminar, modificar y obtener datos de las matrices.

```
class matrizbinaria {
    int filas;
    int columnas;
    int** matriz;
public:
    class no_mem{};
    matrizbinaria(){};
    matrizbinaria(int fila, int columna);
    matrizbinaria(matrizbinaria& m);
    ~matrizbinaria();
    int valor(int fila, int columna);
    int contar_filas();
    int contar_columnas();
    int sumar_fila(int fila);
    int sumar_columna(int columna);
    void poner1(int fila, int columna);
    void poner0(int fila, int columna);
    void mostrar();
    void intercambiar(matrizbinaria& m, int* vals);
    void inicializar();
    void alterar_distribucion(Lista<Lista<int> >& lis);
    bool filas_iguales(int fil1, int fil2);
    bool columnas_iguales(int col1, int col2);
    friend matrizbinaria& operator+(matrizbinaria& m1, matrizbinaria& m2);
    matrizbinaria& operator=(matrizbinaria& m);
    bool operator ==(matrizbinaria& m);
    bool operator !=(matrizbinaria& m);
    friend ostream& operator<<(ostream& os, matrizbinaria& ma);
};
```

11.3.5. Clase Matriz

Esta clase no necesita demasiada explicación ya que es exactamente igual a la del apartado anterior exceptuando los valores que puede almacenar la matriz. En los objetos creados por esta clase se pueden guardar cualquier tipo de valores enteros, no solamente 0 y 1.

```
class matriz {
    int filas;
    int columnas;
    int** matrix;
public:
    class no_mem{};
    matriz(){};
    matriz(int fila, int columna);
    matriz(matriz& m);
    ~matriz();
    int valor(int fila, int columna);
    int contar_filas();
    int contar_columnas();
    void poner_valor(int fila, int columna, int& valor);
    void mostrar();
    void inicializar();
    friend matriz& operator+(matriz& m1, matriz& m2);
    matriz& operator=(matriz& m);
    bool operator ==(matriz& m);
    bool operator !=(matriz& m);
    friend ostream& operator<<(ostream& os, matriz& ma);
};
```

12. Conclusiones y trabajos futuros

Al terminar el proyecto se ha llegado a una serie de conclusiones y se ha pensado en qué trabajos se pueden realizar más adelante. Este apartado enuncia todo esto.

12.1. Conclusiones

Inicialmente se han propuesto un conjunto de objetivos, de los cuales, varios no se han podido lograr debido a una serie de cuestiones, pero los resultados realizados del proyecto han sido de nuestro agrado debido a que se han desarrollado muchos puntos y se ha descartado la utilización de muchas técnicas. El problema que se ha tenido en algún punto ha sido a causa de que se nos escapa alguna cuestión de programación. Ha habido algún atasco a la hora de la creación de código pero finalmente, se ha concluido con éxito.

Este se trata de un programa con conceptos poco investigados, sobre todo en lo relativo a la realización de código, pero estamos contentos con el trabajo obtenido y damos pie a que, quien lo desee, se anime a realizar ampliaciones o mejoras sobre el mismo ya que, en mi opinión, puede llegar a funcionar de forma precisa si se realizan una serie de cambios. El trabajo ha sido interesante debido a la variedad de conceptos que se han expuesto, la cantidad de técnicas de programación que se han utilizado y la variedad de estructuras para almacenar datos que se han creado.

12.2. Trabajos futuros

Como hemos dicho, hay puntos que no han llegado a funcionar para los casos más complejos. Sería una buena opción modificar la función, si es posible, donde está implementado el algoritmo BFS para resolver este problema ya que nosotros lo hemos intentado de varias formas y no hemos podido.

Otro de los estudios futuros, sería el cálculo del isomorfismo mediante otra técnica no utilizada en este trabajo, o usando la herramienta del NAUTY si se llegan a resolver los problemas con los que nos hemos encontrado. Lo bonito de añadir este cálculo al programa sería poder calcular, sin la necesidad de ninguna otra herramienta, los digrafos radiales resultantes eliminando los isomorfos.

En un principio se había comentado el realizar varios cálculos más sobre los digrafos resultantes pero debido a la cantidad de trabajo que se ha tenido

en el resto de objetivos no se han realizado. Sería interesante incluír dichos cálculos en la realización de un trabajo futuro.

Referencias

[1] C.L.Avila, *Análisis y diseño de algoritmos, Backtracking, UNT.*

[2] J.M.Brunat, *Combinatòria i teoria de grafs, Edicions UPC.*

[3] F.Comellas, J.Fábrega, A.Sánchez, O.Serra, *Matemática discreta, Upc.*

[4] J.Gimbert, R.Moreno, J.M.Ribó, M.Valls, *Apropament a la teoria de grafs i als seus algorismes, Udl.*

[5] J.Gimbert, R.Moreno, M.Valls, *Notes sobre combinatòria, Paperkite Editorial.*

[6] R.Guerequeta, A.Vallecillo, *Técnicas de diseño de Algoritmos, Servicio de Publicaciones de la Universidad de Málaga, 1997.*

[7] X.Gort, *Anàlisi, disseny i implementació d'una llibreria per a l'estudi de digrafs excèntrics, TFC, Udl, 2003.*

[8] M.Knor, *A note on radially Moore digraphs, IEEE Trans. Comput. 45 No. 3 (1996), 381-383.*

[9] I.López, *Contribución al estudio de excentricidades en grafos dirigidos, Tesis, Udl, 2008.*

[10] R.Masià, J.Pujol, J.Rifà, M.Villanueva, *Matemàtica Discreta, UOC.*

[11] B.D.McKay, *Nauty User's Guide, Computer Science Department of Australian National University, 2002.*

[12] J.M.Ribó, Introducció a l'algorísmica, Quaderns EUP núm.6, Paperkite Editorial.

[13] S.Pozo, *Curso de C++, manual http: www.conclase.net.*